

Записки по ДАА

Тодор Дуков

Съдържание

1	Въведение в алгоритмите и асимптотичния анализ	3
1.1	Що е то алгоритъм?	3
1.2	Какво означава добро решение?	4
1.3	Как мерим времето и паметта?	4
1.4	Основни дефиниции	5
1.5	Полезни свойства	8
1.6	Задачи	9
2	Анализ на сложността на итеративни алгоритми	10
2.1	Как анализираме един алгоритъм по сложност?	10
2.2	Предимствата и недостатъците на този вид анализ	11
2.3	Сложност по време на някои алгоритми	12
2.4	Задачи	14

Глава 1

Въведение в алгоритмите и асимптотичния анализ

1.1 Що е то алгоритъм?

Алгоритмите се срещат навсякъде около нас:

- рецептите са алгоритми за готвене
- сутрешното приготвяне
- придвижването от точка А до точка В
- търсенето на книга в библиотеката

Въпреки това е трудно да се даде формална дефиниция на това какво точно е алгоритъм. На ниво интуиция, човек може да си мисли, че това просто е някакъв последователен списък от стъпки/инструкции, които човек/машина трябва да изпълни. Други начини човек да си мисли за алгоритмите, са:

- програми – обикновено така се реализират алгоритми
- машини на Тюринг, крайни (стекови) автомати или формални граматики
- частично рекурсивни функции

Един програмист в ежедневието си постоянно пише алгоритми за да решава различни задачи/проблеми. Една задача може да се решава по много начини, някои по-добри от други. Добрият програмист, освен че ще намери решение на проблема, той ще намери най-доброто решение (или поне достатъчно добро за неговите цели).

1.2 Какво означава добро решение?

Хубаво е човек да се води по следните (неизчерпателни) критерии:

- решението трябва да е коректно – ако алгоритъмът работи само през 50% от времето, най-вероятно можем да се справим по-добре
- решението трябва да е бързо – ако алгоритъмът ще завърши работа след като всички звезди са измрели, то той практически не ни върши работа
- решението трябва да заема малко памет – ако алгоритъмът по време на своята работа се нуждае от повече памет, колкото компютърът може да предостави, за нас този алгоритъм е безполезен
- решението трябва да е просто – това е може би най-маловажният критерий от тези, но въпреки това е хубаво когато човек може, да пише чист и разбираем код, който лесно се разширява

За да можем да сравняваме алгоритми в зависимост от това колко големи ресурси (време и памет) използват, трябва първо да можем да “измерваме” тези ресурси.

1.3 Как мерим времето и паметта?

Когато пишем алгоритми, имаме няколко базови инструкции (за които предварително сме се уговорили), които ще наричаме **атомарни инструкции**. Тяхното извикване ще отнеме една единица време. **Време за изпълнение** ще наричаме броят на извикванията на атомарните инструкции по време на изпълнение на програмата. Също така числата и символите ще бъдат нашите **атомарни типове данни**, и ще заемат една единица памет. **Паметта**, която една програма заема, ще наричаме максималния брой на единици от атомарни типове данни по време на изпълнение, без да броим входните данни. Обикновено времето и паметта зависят от размера на подадените входни данни. Това означава, че можем да си мислим за времето и паметта като функции на размера на входа. Подходът, който ще изберем, е да сравняваме функциите за време/памет на различните алгоритми асимптотично. Интересуваме се не толкова от конкретните стойности, а от поведението им, когато размерът на входа клони към безкрайност.

1.4 Основни дефиниции

Множеството от функции, които ще анализираме, е

$$\mathcal{F} = \{f \mid f : \mathbb{R}^{\geq 0} \rightarrow \mathbb{R} \ \& \ (\exists n_0 > 0)(\forall n \geq n_0)(f(n) > 0)\}.$$

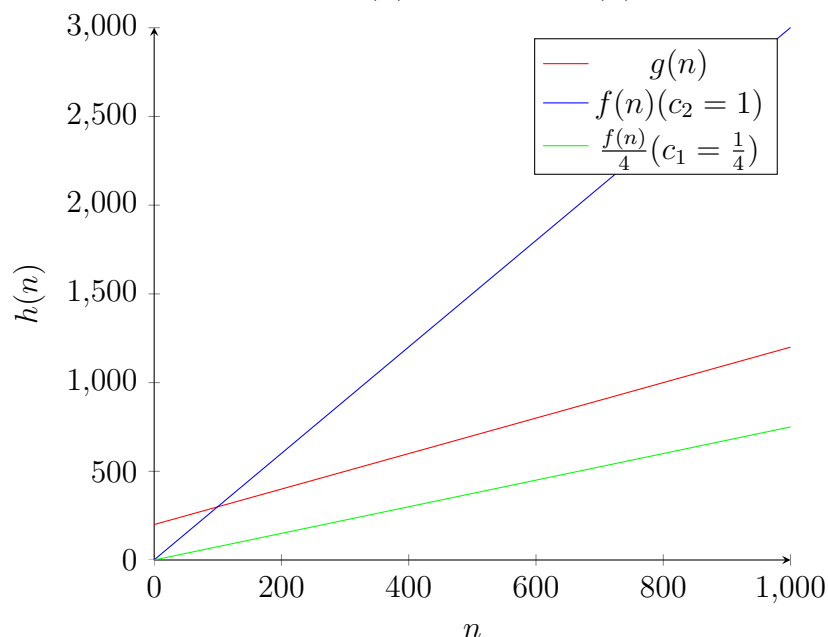
Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$\Theta(f) = \{g \in \mathcal{F} \mid (\exists c_1 > 0)(\exists c_2 > 0) \\ (\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n))\}.$$

Може да тълкуваме $\Theta(f)$ като:

“множеството от функциите, които растат със скоростта на f ”.*

Нека вземем за пример $f(n) = 3n + 1$ и $g(n) = n + 200$:



На картинката се вижда как от един момент нататък, функцията f остава “заклучена“ между $c_1 \cdot g$ и $c_2 \cdot g$. Точно заради това $g \in \Theta(f)$.

Забележка. Вместо да пишем $g \in \Theta(f)$, ще пишем $g = \Theta(f)$ или $g \asymp f$.

*точност до константен множител и константно събираемо

ГЛАВА 1. ВЪВЕДЕНИЕ В АЛГОРИТМИТЕ И АСИМПТОТИЧНИЯ АНАЛИЗ

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

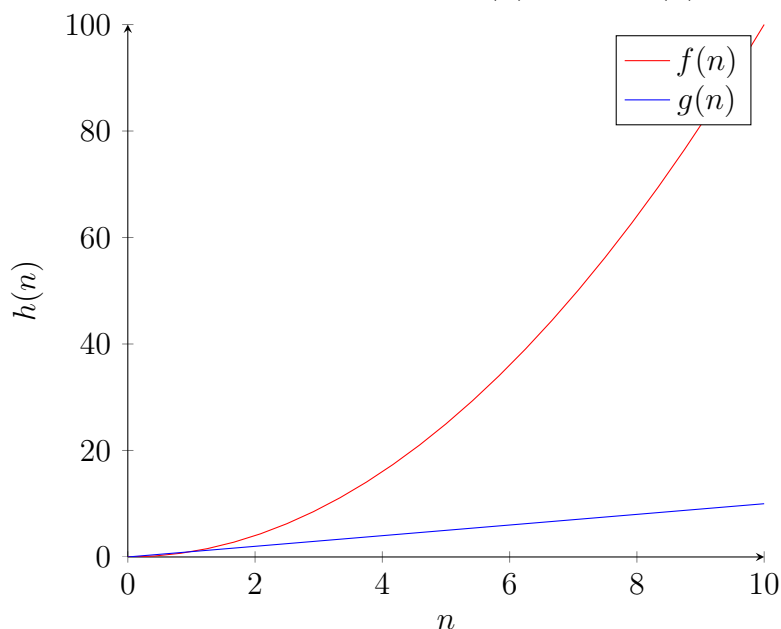
$$O(f) = \{g \in \mathcal{F} \mid (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(g(n) \leq c \cdot f(n))\}.$$

Може да тълкуваме $O(f)$ като:

“множеството от функциите, които не растат* по-бързо от f ”.

Тук заслабваме условията от $\Theta(f)$ като искаме само горната граница.

За пример човек може да вземе $f(n) = n^2$ и $g(n) = n$:



Забележка. Вместо да пишем $g \in O(f)$, ще пишем $g = O(f)$ или $g \preceq f$.

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$o(f) = \{g \in \mathcal{F} \mid (\forall c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(g(n) < c \cdot f(n))\}.$$

Може да тълкуваме $o(f)$ като:

“множеството от функциите, които растат* по-бавно от f ”.

Разликата между $O(f)$ и $o(f)$ е строгото неравенство и универсалният квантор в началото. Лесно се вижда, че $o(f) \subseteq O(f)$. Тук изключваме функциите от същия порядък.

Забележка. Вместо да пишем $g \in o(f)$, ще пишем $g = o(f)$ или $g \prec f$.

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$\Omega(f) = \{g \in \mathcal{F} \mid (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(c \cdot f(n) \leq g(n))\}.$$

Може да тълкуваме $\Omega(f)$ като:

“множеството от функциите, които не растат* по-бавно от f ”.

Това е дуалното множество на $O(f)$.

Забележка. Вместо да пишем $g \in \Omega(f)$, ще пишем $g = \Omega(f)$ или $g \succeq f$.

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$\omega(f) = \{g \in \mathcal{F} \mid (\forall c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(c \cdot f(n) < g(n))\}.$$

Може да тълкуваме $\omega(f)$ като:

“множеството от функциите, които растат* по-бързо от f ”.

Това е дуалното множество на $o(f)$.

Забележка. Вместо да пишем $g \in \omega(f)$, ще пишем $g = \omega(f)$ или $g \succ f$.

Внимание. Не всички функции от \mathcal{F} са сравними по релациите \prec , \preceq или \succ . За пример човек може да вземе функциите $f(n) = n$ и $g(n) = n^{1+\sin(n)}$. Лесно се вижда, че функцията $g(n)$ “плава” между $n^0 = 1$ и n^2 т.е. няма нито как да расте по-бързо, нито как да расте по-бавно.

Въпреки това, тези релации са сравнително хубави.

Твърдение 1.4.1. Следните свойства са в сила:

- \asymp е релация на еквивалентност;
- \prec и \succ са транзитивни и антирефлексивни;
- \preceq и \succeq са транзитивни и рефлексивни.

Доказателството на това твърдение оставяме за упражнение на читателя. То е една елементарна разходка из дефинициите.

1.5 Полезни свойства

Тук ще изброим няколко свойства, които много често се ползват в задачите:

- Нека $f, g \in \mathcal{F}$ и $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = l$ (тук искаме границата да съществува). Тогава:
 - ако $l = 0$, то $f \prec g$
 - ако $l = \infty$, то $f \succ g$
 - в останалите случаи $f \asymp g$
- $f + g \asymp \max\{f, g\}$ за всяко $f, g \in \mathcal{F}$
- $c \cdot f \asymp f$ за всяко $f \in \mathcal{F}$ и $c > 0$
- $f \asymp g \iff f^c \asymp g^c$ за всяко $f, g \in \mathcal{F}$ и $c > 0$
- $O(f) \cap \Omega(f) = \Theta(f)$ за всяко $f \in \mathcal{F}$
- $o(f) \cap \omega(f) = O(f) \cap \Omega(f) = \emptyset$ за всяко $f \in \mathcal{F}$
- $f \prec g \iff g \succ f$ и $f \preceq g \iff g \succeq f$ за всяко $f, g \in \mathcal{F}$
- ако $f \prec g$, то $c^f \prec c^g$ за всяко $f, g \in \mathcal{F}$ и $c > 1$
- ако $\log_c(f) \prec \log_c(g)$, то $f \prec g$ за всяко $f, g \in \mathcal{F}$ и $c > 1$
- ако $c^f \asymp c^g$, то $f \asymp g$ за всяко $f, g \in \mathcal{F}$ и $c > 1$
- ако $f \asymp g$, то $\log_c(f) \asymp \log_c(g)$ за всяко $f, g \in \mathcal{F}$ и $c > 1$
- тъй като $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$, то $\log_a(n) \asymp \log_b(n)$ – вече ще пишем само $\log(n)$ като ще имаме предвид $\log_2(n)$
- $n! \asymp \sqrt{n} \frac{n^n}{e^n}$ – апроксимация на Стирлинг
- $\log(n!) \asymp n \log(n)$
- $\log(n) \prec n^k \prec 2^n \prec n! \prec n^n \prec 2^{n^2}$ за всяко $k \geq 1$

1.6 Задачи

Задача 1.1. Да се сравнят асимптотично следните двойки функции:

1. $f(n) = \log(\log(n))$ и $g(n) = \log(n)$

2. $f(n) = 5n^3$ и $g(n) = n\sqrt{n^9 + n^5}$

3. $f(n) = n5^n$ и $g(n) = n^23^n$

4. $f(n) = n^n$ и $g(n) = 3^{n^2}$

5. $f(n) = 3^{n^2}$ и $g(n) = 2^{n^3}$

Задача 1.2. Да се докаже, че $\sum_{i=0}^n i^k \asymp n^{k+1}$

Задача 1.3. Да се подредят по асимптотично нарастване следните функции:

$$f_1(n) = n^2 \quad f_2(n) = \sqrt{n} \quad f_3(n) = \log^2(n) \quad f_4(n) = \sqrt{\log(n)!}$$

$$f_5(n) = \sum_{k=2}^{\log(n)} \frac{1}{k} \quad f_6(n) = \log(\log(n)) \quad f_7(n) = 2^{2^{\sqrt{n}}} \quad f_8(n) = \binom{\binom{n}{3}}{2}$$

$$f_9(n) = 2^{n^2} \quad f_{10}(n) = 3^{n\sqrt{n}} \quad f_{11}(n) = 2^{\binom{n}{2}} \quad f_{12}(n) = \sum_{k=1}^{n^2} \frac{1}{2^k}$$

Глава 2

Анализ на сложността на итеративни алгоритми

2.1 Как анализираме един алгоритъм по сложност?

Нека започнем с един прост пример:

```
1 find( $A[1 \dots n] \in \text{array}(\mathbb{Z}); v \in \mathbb{Z}$ ):  
2   for  $i \leftarrow 1$  to  $n$ :  
3     if  $A[i] = v$ : return  $i$   
4  
5   return  $-1$ 
```

Да кажем, че искаме да проверим броя на инструкциите, която тази функция ще изпълни, преди да приключи работата си. Точен отговор не може да се даде. В зависимост от това къде се намира v във $A[1 \dots n]$, алгоритъмът може да приключи много бързо или много бавно. Можем да дадем горна и долна граница на бързодействието.

Ако v се намира в началото, то ще сме направили само следните 4 операции:

- да инициализираме променливата i със 0;
- да проверим верността на $i < n$;
- да проверим верността на $A[i] = v$;
- да върнем i т.е. 1.

Нека сега да помислим какво ще стане в най-лошия случай (обикновено от тези ще се интересуваме) – v не участва в $A[1 \dots n]$. Тогава n пъти ще изпълним следните 3 операции:

- проверяваме верността на $i \leq n$;
- проверяваме верността на $A[i] = v$;
- увеличаваме i с 1.

Освен тези $3n$ операции, преди всичко трябва да инициализираме променливата i със 1, да се направи последната проверка на верността на $i \leq n$ (която ще ни изкара от цикъла), и да върнем -1 . Общо излизат $3n + 3$ операции.

Така виждаме, че в зависимост от входните данни, алгоритъмът приключва работа за поне 4 стъпки и най-много $3n + 3$ стъпки. Такъв алгоритъм ще казваме, че има сложност по време $O(n)$. Разбира се, няма да е грешно и да кажем, че алгоритъмът има сложност по време $\Omega(1)$, но това не ни дава никаква информация, защото всеки алгоритъм има такава сложност. Също така, понеже не използваме допълнителни променливи, алгоритъмът ни има константна сложност по памет или сложност по памет $\Theta(1)$.

По-общо казано, се интересуваме от асимптотиката на $T(n)$, където $T(n)$ е броят елементарни инструкции, които алгоритъмът извиква по време на своето изпълнение, при вход с размер n в най-лошият случай.

Тук вход с големина n може да означава различни неща. Ако входът е някакъв масив или множество, то под размер ще разбираме броят на елементи. Ако пък входът е число, то под размер можем да разбираме самата стойност на числото или дължината на двоичния запис.

2.2 Предимствата и недостатъците на този вид анализ

Най-голямото предимство на асимптотичния анализ, е неговата простота. Вместо да влачим някакви константни множители и събираеми, имаме колкото се може по-проста формула, която да описва сложността на нашия алгоритъм. Това дали един алгоритъм работи със две или три стъпки по-бързо/бавно не ни интересува особено много. При много голям вход те ще работят практически еднакво. В някакъв смисъл това ни помага да виждаме по-голямата картинка. Един алгоритъм може да бъде по-бърз от друг, но от по-бърз алгоритъм до по-бърз алгоритъм има голяма разлика.

ГЛАВА 2. АНАЛИЗ НА СЛОЖНОСТТА НА ИТЕРАТИВНИ АЛГОРИТМИ

Нека вземем за пример следната таблица:

n	$\lceil \log_2(n) \rceil$	n	n^2	2^n
1	0	1	1	2
10	4	10	100	1024
100	7	100	10000	число със 31 цифри
10000	13	10000	100000000	число със 3011 цифри
1000000	20	1000000	1000000000000	число със 301030 цифри

Алгоритъм със сложността n^2 ще е по-бавен от алгоритъм със сложност n , обаче скока в бързината е много по-малък от този между 2^n и n^2 .

Този подход обаче си има своите недостатъци. Нека разгледаме два алгоритъма със сложности по време съответно n и $2^{2^{2^{1024}}}$. Ние веднъж ще се втурнем да кажем, че първият алгоритъм е по-лош. Той е с линейна сложност, а вторият алгоритъм има константна сложност. Обаче преди вторият алгоритъм даде отговор, всички звезди ще умрат т.е. няма да доживеем да чуем този отговор. Разбира се, от някъде нататък, за много големи входни данни, първият алгоритъм наистина ще работи по-бавно, но ние никога няма да работим с толкова големи данни. Тогава на практика, първият алгоритъм е по-добър, нищо че асимптотично се води по-лош. Нас това няма да ни интересува в курса по ДАА.

2.3 Сложност по време на някои алгоритми

Нека видим сложността на алгоритъма за сортиране по метода на мехурчето:

```
1 Sort( $A[1 \dots n] \in \text{array}(\mathbb{Z})$ ):  
2   for  $i \leftarrow 1$  to  $n - 1$ :  
3     for  $j \leftarrow 1$  to  $n - i - 1$ :  
4       if  $A[j] > A[j + 1]$ :  
5          $temp \leftarrow A[j]$   
6          $A[j] \leftarrow A[j + 1]$   
7          $A[j + 1] \leftarrow temp$ 
```

В най-лошия случай сложността $T(n)$ на функцията **Sort** е следната:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i-1} 1 = \sum_{i=1}^{n-1} (n-i-1) = (n-2) + (n-3) + \dots + 0 = \frac{(n-2)(n-1)}{2} \asymp n^2.$$

ГЛАВА 2. АНАЛИЗ НА СЛОЖНОСТТА НА ИТЕРАТИВНИ АЛГОРИТМИ

По принцип $T(n)$ трябва да е сума от 4, а не от 1, но такъв константен брой операции, дори и приложени неконстантен брой пъти, не влияят на асимптотичното поведение.

Нека сега разгледаме следният алгоритъм за степенуване:

```
1 Exp( $x, y \in \mathbb{N}$ ):  
2    $res \leftarrow 1$   
3  
4   while  $y > 0$ :  
5       if  $y \equiv 1 \pmod{2}$ :  
6            $res \leftarrow res \cdot x$   
7  
8        $y \leftarrow \frac{y}{2}$   
9        $x \leftarrow x^2$   
10  
11  return  $res$ 
```

Той се възползва от простата идея, че за да сметнем да кажем 3^8 , можем вместо 8 пъти да умножаваме числото 3, да представим 3^8 като $3^4 \cdot 3^4$. Тогава 3^4 можем да сметнем веднъж, и да го умножим със себе си. Пак можем да представим 3^4 като $3^2 \cdot 3^2$ и да пресметнем 3^2 само веднъж и да го умножим със себе си. Така при по-голяма стойност на y си спестяваме много работа. С уговорката, че умножението е атомарна операция, сложността по време $T(n)$ (n е стойността на y) на функцията Exp е следната:

$$T(n) = \sum_{\substack{i=n \\ i \leftarrow \frac{i}{2}}}^1 1 = \underbrace{1 + \dots + 1}_{\substack{\text{колкото пъти} \\ \text{можем да} \\ \text{делим целочислено} \\ n \text{ на } 2 \text{ преди} \\ \text{да получим } 0}} = \underbrace{1 + \dots + 1}_{\text{около } \log(n) \text{ пъти}} \asymp \log(n).$$

2.4 Задачи

Задача 2.1. Да се определи сложността по време за функцията:

```

1  $F_1(n \in \mathbb{N})$  :
2    $s \leftarrow 0$ 
3
4   for  $i \leftarrow 1$  to  $n$ :
5     for  $j \leftarrow 1$ ;  $j \leq i$ ;  $j \leftarrow 2j$ :
6        $s \leftarrow s + ij$ 
7
8   return  $s$ 
```

Задача 2.2. Да се определи сложността по време за функцията:

```

1  $F_2(n \in \mathbb{N})$  :
2    $s \leftarrow 0$ 
3
4   for  $i \leftarrow 1$ ;  $i^2 \leq n$ ;  $i \leftarrow i + 3$ :
5      $s \leftarrow s + F_1(n)$ 
6
7   return  $s$ 
```

Задача 2.3. Да се определи сложността по време за функцията:

```

1  $F_3(n \in \mathbb{N})$  :
2    $s \leftarrow 0$ 
3
4   for  $i \leftarrow 1$ ;  $i \leq n^2$ ; inc( $i$ ):
5     for  $j \leftarrow 1$ ;  $j \leq 2i$ ; inc( $j$ ):
6       if  $j \equiv 0 \pmod{2}$ :
7          $s \leftarrow s + F_1(n)$ 
8       else:
9          $s \leftarrow s + F_2(n)$ 
10
11  return  $s$ 
```